Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability[*]

Andrea Bastoni[†] Björn B. Brandenburg[‡] James H. Anderson[‡]

Abstract

A job that is impeded by a preemption or migration incurs additional cache misses when it resumes execution due to a loss of cache affinity. While often regarded negligible in scheduling-theoretic work, such cache-related delays must be accounted for when comparing scheduling algorithms in real systems. Two empirical methods to approximate cache-related preemption and migration delays on actual hardware are proposed, and a case study reporting measured average- and worst-case overheads on a 24-core Intel system with a hierarchy of shared caches is presented. The widespread belief that migrations are always more costly than preemptions is refuted by the observed results. Additionally, an experiment design for schedulability studies that allows algorithms to be compared objectively under consideration of cache-related delays is presented.

1 Introduction

A controversial topic with regard to the choice of scheduler in multiprocessor real-time systems—both within academia and among practitioners—is the relative impact of *cache-related preemption and migration delay* (CPMD), *i.e.*, the delay that a preempted job incurs due to a loss of cache affinity after resuming execution.

Traditionally, migrations are considered to be a source of unacceptable overhead, and thus implementors tend to favor *partitioning* to avoid migrations altogether. This view, however, conflicts with scheduling-theoretic advances from the last decade that show that there exist *global* schedulers, which allow jobs to migrate freely, that are provably superior to partitioning if overheads (including CPMD) are negligible [5, 17, 19, 24, 33]. In contrast, if CPMD is deemed to be significant (which, of course, it is in practice), then one can easily *construct* scenarios in which global scheduling is not a viable alternative by assuming prohibitively high migration costs. However, how realistic are such scenarios?

Clearly, a meaningful comparison of schedulers requires CPMD to be taken into account, yet arbitrarily choosing delays is of only little benefit. The problem is compounded by the difficulty of measuring CPMD [9], which can only be observed indirectly and is heavily dependent on the *working set size* (WSS) of each task, in contrast to other OS delays such as scheduling overhead [8, 9].

This raises three central questions:

1. How can CPMD be estimated empirically when evaluating algorithms for a specific platform?
2. What are reasonable values to assume for CPMD when evaluating (newly-proposed) algorithms in scheduling-theoretic work?
3. How can schedulers be evaluated without accidentally introducing a bias towards a particular WSS?

In particular, we are interested in *simple*, yet effective, methods that can be realistically performed as part of scheduling research and by practitioners during early design phases (*i.e.*, when selecting platforms and algorithms).

Contributions. In this paper, we propose two approaches to measure CPMD: a "schedule-sensitive method" that can measure scheduler-dependent cache effects (Sec. 3.1), and a "synthetic method" that can be used to quickly record a large number of samples (Sec. 3.2). (A preliminary version of the "schedule-sensitive method" was previously used—but not described in detail—in [9].)

To demonstrate the efficacy of our approaches, we report average and maximum CPMD for various WSSs on a current 24-core Intel platform with hierarchical caches (Sec. 4). Perhaps surprisingly, in a system under load, *migrations were found to not cause significantly more delay than preemptions* (Sec. 4.2). In particular, our results show that CPMD is **(i)** in excess of one millisecond for working set sizes exceeding 256 KB (Fig. 2(a)), **(ii)** ill-defined if there is heavy contention for shared caches (Fig. 2(c)), **(iii)** strongly dependent on the preemption length (Fig. 3), and **(iv)** not dependent on task set size (Fig. 4).

Finally, we discuss how CPMD can be integrated into large-scale schedulability studies without rendering the results dependent on particular WSS assumptions (Sec. 5).

Related work. Accurately assessing cache-related delays is a classical component of *worst-case execution time* (WCET) analysis [42], in which an upper bound on the

[*]Work supported by AT&T, IBM, and Sun Corps.; NSF grants CNS 0834270, CNS 0834132, and CNS 0615197; ARO grant W911NF-09-1-0535; and AFOSR grant FA 9550-09-1-0549.

[†]SPRG, University of Rome "Tor Vergata"

[‡]Dept. of Computer Science, U. of North Carolina at Chapel Hill

maximum resource requirements of a real-time task is derived *a priori* based on control- and data-flow analysis. Unfortunately, predicting cache contents and hit rates is notoriously difficult: even though there has been some initial success in bounding *cache-related preemption delays* (CPDs) caused by simple data [31] and instruction caches [37], analytically determining preemption costs *on uniprocessors with private caches* is still generally considered to be an open problem [42]. Thus, on multicore platforms with a *complex hierarchy of shared caches*, we must—at least for now—resort to empirical approximation. However, given recent advances in bounding migration delays [21] and analyzing interference due to shared caches [16, 43], we expect multicore WCET analysis to be developed eventually.

Trace-driven memory simulation [41], in which memory reference traces collected from actual program executions are interpreted with a cache simulator, has been applied to count cache misses after context switches [29, 36]. Using traces from throughput-oriented workloads, Mogul and Borg [29] estimated CPDs to lie within $10\mu s$ to $400\mu s$ on an early '90s RISC-like uniprocessor with caches ranging in size from 64 to 2048 kilobytes. In work on real-time systems, Stärner and Asplund [36] used trace-driven memory simulation to study CPDs in benchmark tasks on a MIPS-like uniprocessor with small caches. As the simulation environment is fully controlled, this method allows cache effects to be studied in great detail, but it is also limited by its reliance on accurate architectural models (which may not always be available) and representative memory traces (which are difficult to collect due to complex instrumentation requirements).

Several probabilistic models have been proposed to predict expected cache misses on uniprocessors [1, 27, 39]. In the context of evaluating (hard) real-time schedulers, such models apply only to a limited extent because it is difficult to extract bounds on the worst-case number of cache misses. Further, they rely on task parameters that are difficult to obtain or predict (e.g., cache access profiles [27]), and do not predict cache misses after migrations.

Closely related to our approach are several recent CPD microbenchmarks [18, 25, 40]. Li *et al.* [25] measured the cost of switching between two processes that alternate between accessing a data array and communicating via a pipe on an Intel Xeon processor with a 512kB L2 cache, and found that average case CPDs can range from around $100\mu s$ to $1500\mu s$, depending on array size and access pattern. In the context of real-time systems, Li *et al.*'s experimental setup is limited because it can only estimate average-case, but not worst-case, delays. David *et al.* [18] measured preemption delays in Linux kernel threads on an embedded ARM processor with comparably small caches and observed CPDs in the range of $60\mu s$ to $120\mu s$. Tsafrir [40] investigated the special case in which the scheduled job is not preempted, but cache contents are perturbed by periodic clock interrupts, and found that slowdowns vary heavily among workloads. None of the cited empirical studies considered job migrations.

Once bounds on cache-related delays are known for a given task set, they must be accounted for during schedulability analysis. This is typically accomplished by inflating task parameters to reflect the time lost to reloading cache contents. Straightforward methods are known for common uniprocessor schedulers [6, 11, 22] and have also been derived for global schedulers [19, 34]; we use this approach in Sec 5. A limitation of these methods is that each preemption between any two tasks is assumed to cause maximal delay, an assumption that is likely unnecessarily pessimistic. More advanced methods that yield tighter bounds by analyzing per-task cache use and the instant at which each preemption occurs have been developed for static-priority uniprocessor schedulers [23, 30, 37]. However, similar to WCET analysis, these methods have not yet been generalized to multiprocessors since they require useful cache contents to be predicted accurately. Stamatescu *et al.* [35] propose including average memory access costs in specific analysis, but do not report measured costs.

Several research directions orthogonal to this paper are concerned with avoiding, or at least reducing, cache-related delays in multiprocessor real-time systems. On an architectural level, Sarkar *et al.* [32] have proposed a scheduler-controlled cache management scheme that enables cache contents to be transferred in bulk instead of relying on normal cache-consistency updates. This can be employed to lessen migration costs by transferring useful cache contents before a migrated job resumes [32]. Likewise, Suhendra and Mitra [38] have considered cache locking and partitioning policies to isolate real-time tasks from timing interference due to shared caches. While promising, neither technique is supported in current multicore architectures.

In work on real-time scheduling, numerous recently-proposed schedulers aim to balance the advantages of partitioning and global scheduling by reducing the number of migrations. Such hybrid approaches can be classified into two families: in *semi-partitioned* schedulers [2], most jobs are fixed to processors and only few migrate, whereas in *clustered* schedulers [4, 13], all jobs may migrate, but only among processors that share a cache. Going a step further, *cache-aware* schedulers [12, 20], which make shared caches an explicitly-managed resource, have been proposed to both prevent interference in hard real-time systems [20] and to encourage data reuse in soft real-time systems [12].

Work on hybrid schedulers makes strong assumptions on the relative costs of migrations and preemptions—to fairly evaluate the merits of said approaches thus requires sound estimates of cache-related delays. We detail our methods for obtaining such estimates in Sec. 3, after briefly summarizing required background next.

2 Background

CPMD is a characteristic of modern processors and independent of any particular task model. Given our interest in real-time systems, our focus is the classic *sporadic task model* [15, 26, 28] in which a workload is specified as a collection of *tasks*. Each task T_i is characterized by a *WCET* e_i and a *minimum inter-arrival time* or *period* p_i, and releases a *job* for execution at most once every p_i time units.

Such a job J is *preempted* if its execution is temporarily paused before it is completed, *e.g.*, in favor of another job with higher priority. Suppose J is preempted at time t_p on processor P and resumes execution at time t_r on processor R. J is said to have incurred a *preemption* if $P = R$, and a *migration* otherwise. In either case, we call $t_r - t_p$ the *preemption length*. A job may be preempted multiple times.

Scheduling. Let m denote the number of processors. There are two fundamental approaches to scheduling sporadic tasks on multiprocessors [15]: with *global* scheduling, processors are scheduled by selecting jobs from a single, shared queue, whereas with *partitioned* scheduling, each processor has a private queue and is scheduled independently using a uniprocessor scheduling policy. *Clustered* scheduling [4, 13] is a generalization of both approaches: tasks are partitioned onto m/c clusters of c processors each, which are then scheduled globally (with respect to the processors in each cluster). We use the *earliest-deadline-first* (EDF) policy in each category, *i.e.*, in this paper, we consider *partitioned EDF* (P-EDF, $c = 1$), *clustered EDF* (C-EDF, $1 < c < m$), and *global EDF* (G-EDF, $c = m$).

Caches. Modern processors employ a hierarchy of fast *cache memories* that contain recently-accessed instructions and operands to alleviate high off-chip memory latencies. Caches are organized in layers (or levels), where the fastest (and usually smallest) caches are denoted *level-1* (L1) caches, with deeper caches (L2, L3, *etc.*) being successively larger and slower. A cache contains either instructions or data, and may contain both if it is *unified*. In multiprocessors, *shared* caches serve multiple processors, in contrast to *private* caches, which serve only one.

Caches operate on blocks of consecutive addresses called *cache lines* with common sizes ranging from 8 to 128 bytes. In *direct mapped* caches, each cache line may only reside in one specific location in the cache. In *fully associative* caches, each cache line may reside at any location in the cache. In practice, most caches are *set associative*, wherein each line may reside at a fixed number of locations.

The set of cache lines accessed by a job is called the *working set* (WS) of the job; workloads are often characterized by their *working set sizes* (WSSs). A cache line present in a cache is *useful* if it is going to be accessed again. If a job references a cache line that cannot be found in a level-X cache, then it suffers a *level-X cache miss*. This can occur for several reasons. *Compulsory misses* are triggered the first time a cache line is referenced. *Capacity misses* result if the WSS of the job exceeds the size of the cache. Further, in direct mapped and set associative caches, *conflict misses* arise if useful cache lines were evicted to accommodate mapping constraints of other cache lines. A shared cache must exceed the combined WS of all jobs accessing it, otherwise, frequent capacity and conflict misses may arise due to *cache interference*. Jobs that incur frequent level-X capacity and conflict misses even if executing in isolation are said to be *thrashing* the level-X cache.

Cache affinity describes the effect that a job's overall cache miss rate tends to decrease with increasing execution time (unless it thrashes all cache levels)—after an initial burst of compulsory misses, most useful cache lines have been brought into a cache and do not cause further misses. This explains CPD: when a job resumes execution after a preemption, it is likely to suffer additional capacity and conflict misses as the cache was perturbed [27]. Migrations may further cause affinity for some levels to be lost completely (depending on cache sharing), thus adding compulsory misses to the penalty.

A job's memory references are *cache-warm* after cache affinity has been established; conversely, *cache-cold* references imply a lack of cache affinity.

In this paper, we restrict our focus to *cache-consistent* shared-memory machines: when updating a cache line that is present in multiple caches, inconsistencies are avoided by a *cache consistency protocol*, which either invalidates outdated copies or propagates the new value.

Schedulability. In a *hard real-time system*, each job must complete by its specified deadline, whereas bounded deadline tardiness is permissible in a *soft real-time system* [19]. In the design of a real-time system, a validation procedure—or *schedulability test*—must be used to determine *a priori* whether all timing constraints will be met.

As discussed in Sec. 1, current WCET analysis is limited to yield bounds assuming non-preemptive execution [42, 43]. Hence, schedulability tests must be augmented to reflect system overheads such as CPMD [6, 11, 19, 22, 28, 34]. In particular, under each of the aforementioned EDF variants, it is sufficient to inflate each task's execution cost by the maximum delay caused by one preemption or migration [19, 28]. Formally, let D_c denote a bound on the maximum CPMD incurred by any job, and let, for each task T_i, $e'_i = e_i + D_c$ denote the inflated execution cost: all timing constraints will be met if the task system passes a schedulability test assuming an execution cost of e'_i for each T_i [19, 28]. Generally speaking, CPMD can be factored into execution costs using similar scheduler-specific formulas as long as the maximum number of preemptions incurred or caused by a job can be bounded.

In practice, additional delay sources such as scheduling overheads [8, 9, 14] and interrupt interference [8, 10] must also be taken into account using similar methods, but such

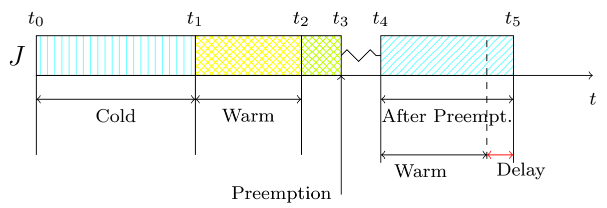


Figure 1: Cache-delay measurement.

considerations are beyond the scope of this paper—our focus is to empirically approximate D_c in real systems.

3 Measuring Cache-Related Delays

Recall that a job is delayed after a preemption or a migration due to a (partial) loss of cache affinity. To measure such delays, we consider jobs that access their WS as illustrated in Fig. 1: a job J starts executing cache-cold at time t_0 and experiences compulsory misses until time t_1, when its WS is completely loaded into cache. After t_1, each subsequent memory reference by J is cache-warm. At time t_2, the job has successfully referenced its *entire* WS in a cache-warm context. From t_2 onward, the job repeatedly accesses single words of its WS (to maintain cache affinity) and checks after each access if a preemption or migration has occurred. Suppose that the job is preempted at time t_3 and not scheduled until time t_4. As J lost cache affinity during the interval $[t_3, t_4]$, the length of the interval $[t_4, t_5]$ (*i.e.*, the time needed to reference again its *entire* WS) reflects the time lost to additional cache misses.

Let d_c denote the cache-related delay suffered by J. After the WS has been fully accessed for the third time (at time t_5), d_c is given by the difference $d_c = (t_5 - t_4) - (t_2 - t_1)$.[1] After collecting a trace $d_{c,0}, d_{c,1}, \ldots, d_{c,k}$ from a sufficiently large number of jobs k, $\max_l\{d_{c,l}\}$ can be used to approximate D_c (recall that D_c is a bound on the maximum CPMD incurred by any job). Similarly, average delay and standard deviation can be readily computed during off-line analysis.

On multiprocessors with a hierarchy of shared caches, migrations are categorized according to the level of cache affinity that is preserved (*e.g.*, a job migration between two processors sharing an L2 cache is an *L2-migration*). A *memory migration* does not preserve any level of cache affinity. Migrations can be identified by recording at time t_3 the processor P on which J was executing and at time t_4 the processor R on which J resumes execution.

Each sample $d_{c,l}$ can be obtained either directly or indirectly. A low-overhead clock device can be used to directly measure the WS access times $[t_1, t_2]$ and $[t_4, t_5]$, which immediately yield d_c. Alternatively, some platforms include *hardware performance counters* that can be used to indirectly measure d_c by recording the number of cache misses. The number of cache misses experienced in each interval is then multiplied by the time needed to service a single cache miss. In this paper, we focus on the direct measure of WS access times, as reliable and precise clock devices are present on virtually all (embedded) platforms. In contrast, the availability of suitable performance counters varies greatly among platforms.

Cache-related preemption and migration delays clearly depend on the WSS of a job and possibly on the scheduling policy [9] and on the task set size (TSS). Hence, to detect such dependencies (if any), each trace $d_{c,0}, d_{c,1}, \ldots, d_{c,k}$ should ideally be collected on-line, *i.e.*, as part of a task set that is executing under the scheduler that is being evaluated *without altering the implemented policy*. We next describe a method that realizes this idea.

3.1 Schedule-Sensitive Method

With this method, d_c samples are recorded on-line while scheduling a proper task set under the algorithm of interest. Performing these measurements without changing the regular scheduling of a task set poses the question of how to efficiently distinguish between a cold, warm, and post-preemption (or migration — *post-pm*) WS access. In particular, detecting a post-pm WS access is subtle, as jobs running under OSs with address space separation (*e.g.*, Linux) are generally not aware of being preempted or migrated. Solving this issue requires a low-overhead mechanism that allows the kernel to inform a job of every preemption and migration. Note that the schedule-sensitive method crucially depends on the presence of such a mechanism (a suitable implementation is presented in Sec. 3.3 below).

Delays should be recorded by executing test cases with a wide range of TSSs and WSSs. This likely results in traces with a variable number of valid samples. To obtain an unbiased estimator for the maximum delay, the same number of samples should be used in the analysis of each trace. In practice, this implies that only (the first) k_{min} from each trace can be used, where k_{min} is the minimum number of valid samples among all traces.

Since samples are collected from a valid schedule, the advantage of this method is that it can identify dependencies (if any) of CPMD on scheduling decisions and on the number of tasks. However, this implies that it is not possible to control *when* a preemption or a migration will happen, since these decisions depend exclusively on the scheduling algorithm (which is not altered). Therefore, the vast majority of the collected samples are likely invalid, *e.g.*, a job may not be preempted at all or may be preempted prematurely, and only samples from jobs that execute exactly as shown in Fig. 1 can be used in the analysis. Thus, large traces are required to obtain few samples. Worse, for a given scheduling algorithm, not all combinations of WSS and TSS may be

[1]The interval $[t_2, t_3]$ is not reflected in d_c since jobs are simply waiting to be preempted while maintaining cache affinity during this interval.

able to produce the execution pattern needed in the analysis (*e.g.*, this is the case with G-EDF, as discussed in Sec. 4).

Hence, we developed a second method that achieves finer control over the measurement process by artificially triggering preemptions and migrations of a single task.

3.2 Synthetic Method

In this approach, CPMD measures are collected by a single task that repeatedly accesses working sets of different sizes. The task is assigned the highest priority and therefore it cannot be preempted by other tasks.

In contrast to the schedule-sensitive method, preemptions and migrations are explicitly triggered in the synthetic method. In particular, the destination core and the preemption length are chosen randomly (preemptions arise if the same core is chosen twice in a row). In order to trigger preemptions, L2-migrations, L3-migrations, *etc.* with the same frequency (and thus to obtain an equal number of samples), proper probabilities must be assigned to each core. Furthermore, as the task execution is tightly controlled, post-pm WS accesses do not need to be detected, and no kernel interaction is needed.

The synthetic method avoids the major drawback of the previous approach, as it generates only valid post-pm data samples. This allows a statistically meaningful number of samples to be obtained rapidly. However, as preemption and migration scheduling decision are externally imposed, this methodology cannot determine possible dependencies of CPMD on scheduling decisions or on the TSS.

3.3 Implementation Concerns

Both methods were implemented using LITMUS$^{\mathrm{RT}}$, a real-time Linux extension developed at UNC [14]. The current version of LITMUS$^{\mathrm{RT}}$ is based on Linux 2.6.32.

Precise time measures of WS access times were obtained on the x86 Intel platform used in our experiments by means of the *time-stamp counter* (TSC), a per-core counter that can be used as high-resolution clock device. The direct measure of CPMD on a multiprocessor platform should take into account the imperfect alignment of per-processor clock devices (*clock skew*). Clock skew errors can be avoided if WS access times are evaluated only based on samples obtained on the same processor (*e.g.*, in Fig. 1, t_1 and t_2 should be measured on the same processor, which may differ from the processor where t_4 and t_5 are measured). In most OSs, time interval measurements can be further perturbed by interrupt handling. These disturbances can be avoided by disabling interrupts while measuring WS access times. Although this does not prevent *non-maskable interrupts* (NMIs) from being serviced, NMIs are infrequent events that likely only have a minor impact on CPMD approximations. We note, however, that our methodology currently cannot detect interference from NMIs.

Disabling interrupts under the schedule-sensitive method is a tradeoff between accuracy and the rate at which samples are collected. On the one hand, disabling interrupts increases the number of valid samples, but on the other hand, it implicitly alters the scheduling policy by introducing non-preemptive sections. We chose to disable interrupts to reduce the length of the experiments.

Within LITMUS$^{\mathrm{RT}}$, we implemented the low-overhead kernelspace–userspace communication mechanism required by the schedule-sensitive method by sharing a single per-task memory page (the *control page*) between the kernel and each task. A task can infer whether it has been preempted or migrated based on the control page: when it is selected for execution, the kernel updates the task's control page by increasing a preemption counter and the job sequence number, storing the preemption length, and recording on which core the task will start its execution.

4 Case Study

To verify and compare results of the two presented methods, we measured cache-related preemption and migration delays using both methodologies on an Intel Xeon L7455. The L7455 is a 24-core 64-bit uniform memory access (UMA) machine with four physical sockets. Each socket contains six cores running at 2.13 GHz. All cores in a socket share a unified 12-way set associative 12 MB L3 cache, while groups of two cores each share a unified 12-way set associative 3 MB L2 cache. Every core also includes an 8-way set associative 32 KB L1 data cache and an identical L1 instruction cache. All caches have a line size of 64 bytes.

4.1 Experimental Setup

We used the G-EDF algorithm to measure CPMD with the schedule-sensitive method, but we emphasize that the method can be applied to other algorithms as well. For this method, we measured the system behavior of periodic task sets consisting of 25 to 250 tasks in steps of variable sizes (from 20 to 30, with smaller steps where we desired a higher resolution). Task WSSs were varied over $\{4, 32, 64, \ldots, 2048\}$ KB. Per-WSS write ratios of 1/2 and 1/4 were assessed.[2] For each WSS and TSS, we measured ten randomly-generated task sets using parameter ranges from [8, 9]. Each task set was traced for 60 seconds and each experiment was carried out once in an otherwise idle system and once in a system loaded with best-effort cache-polluter tasks. Each of these tasks was statically assigned to a core and continuously thrashed the L1, L2, and L3 caches

[2]In preliminary tests with different write ratios, 1/2 and 1/4 showed the highest worst-case overheads, with 1/4 performing slightly worse. All write ratios are given with respect to individual words, not cache lines. There are eight words in each cache line, thus each task updated every cache line in its WS multiple times. Tests with write ratios lower than 1/8, under which some cache lines are only read, exhibited reduced overheads.

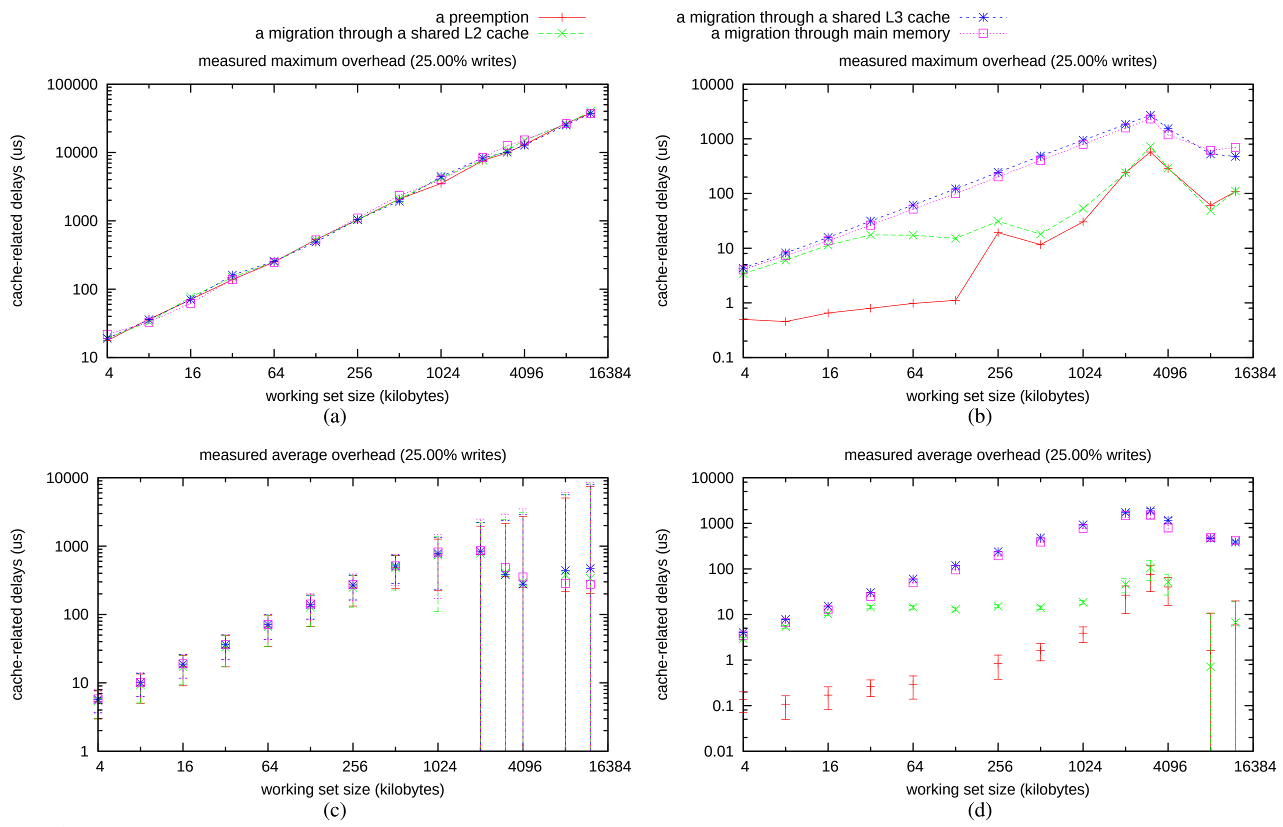


Figure 2: CPMD approximations obtained with the synthetic method. The graphs show maximum and average CPMD (in μs) for preemptions and different types of migrations as a function of WSS (in KB). **(a)** Worst-case delay under load. **(b)** Worst-case delay in an idle system. **(c)** Average-case delay under load. **(d)** Average-case delay in an idle system. The error bars indicate one standard deviation.

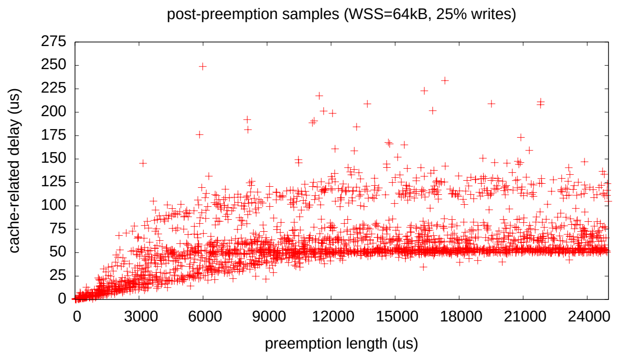
by accessing large arrays. In total, more than 50 GB of trace data with 600 million overhead samples were obtained during more than 24 hours of tracing.

We used a single SCHED_FIFO task running at the highest priority to measure CPMD with the synthetic method. The WSS was chosen from $\{4, 8, 16, \ldots, 8192\}$ KB. We further tested WSSs of 3 and 12 MB, as they correspond to the sizes of the L2 and L3 cache respectively. In these experiments, several per-WSS write ratios were used. In particular, we considered write ratios ranging over $\{0, 1/128, 1/64, 1/16, 1/4, 1/2, 1\}$. For each WSS we ran the test program until 5,000 valid after-pm samples were collected (for each preemption/migration category). Preemption lengths were uniformly distributed in $[0ms, 50ms]$. As with the schedule-sensitive method, experiments were repeated in an idle system and in a system loaded with best-effort cache-polluter tasks. More than 3.5 million *valid* samples were obtained during more than 50 hours of tracing.

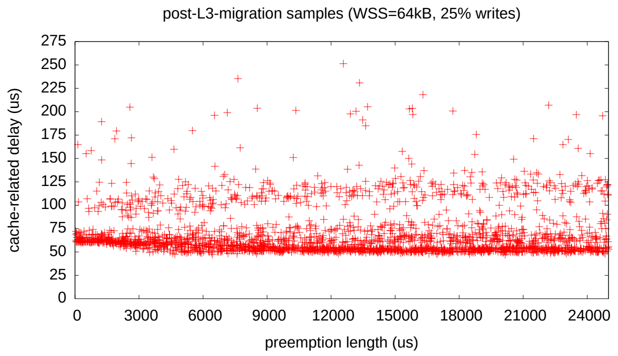
4.2 Results

Fig. 2 shows preemption and migration delays that were measured using the synthetic method (the data is given numerically in Appendix A). Each inset indicates CPMD values for preemptions and all different kinds of migrations (L2, L3, memory) as a function of WSS, assuming a write ratio of 1/4. The first column of the figure (insets (a,c)) gives delays obtained when the system was loaded with cache-polluter tasks, while the second column (insets (b,d)) gives results that were recorded in an otherwise idle system. The first row of the figure presents worst-case overheads, and the second row shows average overheads; the error bars depict one standard deviation. Both axes are in logarithmic scale. Note that these graphs display the difference between a post-pm and and a cache-warm WS access. Declining trends with increasing WSSs (insets (b,c,d)) thus indicate that the cache-warm WS access cost is increasing more rapidly than the post-pm WS access.

Observation 1. The predictability of overhead measures is heavily influenced by the size of L1 and L2 caches. This can be seen in inset (c): as the WSS approaches the size of the L2 cache (3072 KB, shared among 2 cores), the standard deviation of average delays becomes very large (the same magnitude of the measure itself) and therefore overhead estimates are very imprecise. This unpredictability arises because jobs with large WSSs suffer frequent L2- and L3-cache misses in a system under load due to thrashing and cache interference, and thus become exposed to memory bus contention. Due to the thrashing cache-polluter tasks, bus access times are highly unpredictable and L3 cache interference is very pronounced. In fact, our traces show that



(a)



(b)

Figure 3: Scatter plot of observed d_c samples vs. preemption length in a system under load. **(a)** Samples recorded after a preemption. **(b)** Samples recorded after an L3-migration. The plots have been truncated at $25ms$; there are no trends apparent in the range from $25ms$ to $50ms$.

jobs frequently incur "negative CPMD" in such cases because the "cache-warm" access itself is strongly interfered with. This implies that, from the point of view of schedulability analysis, CPMD is not well-defined for such WSSs, since a true WCET must account for worst-case cache interference and thus is already more pessimistic than CPMD, *i.e.*, actual CPMD effects are likely negligible compared to the required bounds on worst-case interference.

Observation 2. In a system under load, there are *no substantial differences* between preemption and migration costs, both in the case of worst-case (inset (a)) and average-case (inset (c)) delays. When a job is preempted or migrated in the presence of heavy background activity, its cache lines are likely evicted quickly from all caches and thus virtually every post-pm access reflects the high overhead of refetching the entire WS from memory. Inset (a) shows that, in a system under load, the worst-case delay for a 256 KB WSS exceeds $1ms$, while the cost for a 1024 KB WSS is around $5ms$. Average-case delays (inset (c)) are much lower, but still around $1ms$ for a 1024 KB WSS.

Observation 3. In an idle system, preemptions always cause less delay than migrations, whereas L3- and memory migrations have comparable costs. This behavior can be observed in insets (b,d). In particular, if the WS fits into the L1 cache (32 KB), then preemptions are negligible (around $1\mu s$), while they have a cost that is comparable with that of an L2 migration when the WSS approaches the size of the L2 cache (still, they remain less than $1ms$). Inset (d) clearly shows that L2-migrations cause less delay than L3-migrations for WSSs that exceed the L1-cache size (about $10\mu s$ for WSSs between 32 and 1024 KB). In contrast, L3- and memory migrations have comparable costs, with a maximum around $3ms$ with 3072 KB WSS (inset (b)). Interestingly, memory migrations cause slightly less delay than L3 cache migrations. As detailed below, this is most likely related to the cache consistency protocol.

Observation 4. The magnitude of CPMD is strongly related to preemption length (unless cache affinity is lost completely, *i.e.*, in the case of memory migrations). This trend is apparent from the plots displayed in Fig. 3. Inset (a) shows individual preemption delay measurements arranged by increasing preemption length, inset (b) similarly shows L3-migration delay. The samples were collected using the synthetic method with a 64 KB WSS and a write ratio of 1/4 in a system under load (similar trends were observed with all WSSs $\leq$ 3072 KB). In both insets, CPMD converges to around $50\mu s$ for preemption lengths exceeding $10ms$. This value is the delay experienced by a job when its WSS is reloaded entirely from memory.[3] In contrast, for preemption lengths ranging in $[0ms, 10ms]$, average preemption delay increases with preemption length (inset (a)), while L3-migrations (in the range $[0ms, 5ms]$) progressively decrease in magnitude (inset (b)). The observed L3-migration trend is due to the cache consistency protocol: if a job resumes quickly after being migrated, parts of its WS are still present in previously-used caches and thus need to be evicted. In fact, if the job does not update its WS (*i.e.*, if the write ratio is 0), then the trend is not present.

Observation 5. Preemption and migration delays do not depend significantly on the task set size. This can be observed in Fig. 4, which depicts worst-case delay for the schedule-sensitive method in a system under load as function of the TSS. The plot indicates CPMD for preemptions and all migration types for WSSs of 1024, 512 and 256 KB (from top to bottom).

Note that Fig. 4 is restricted to TSSs from 75 to 250 because, under G-EDF, only few task migrations occur for small TSSs. Thus, the number of collected valid delays for small TSSs is not statistically meaningful.

Furthermore, Fig. 4 shows that worst-case preemption and migrations delays for the same WSS have comparable magnitudes, thus confirming that, in a system under load, preemption and migration costs do not differ substantially (recall Fig. 2(a) and Observation 2).

[3]Due to space limitations, plots for memory and L2-migrations are not shown. L2-migrations reveal a trend that is similar to the preemption case, while memory migrations do not show a trend (samples are clustered around $50\mu s$ delay regardless of preemption length).

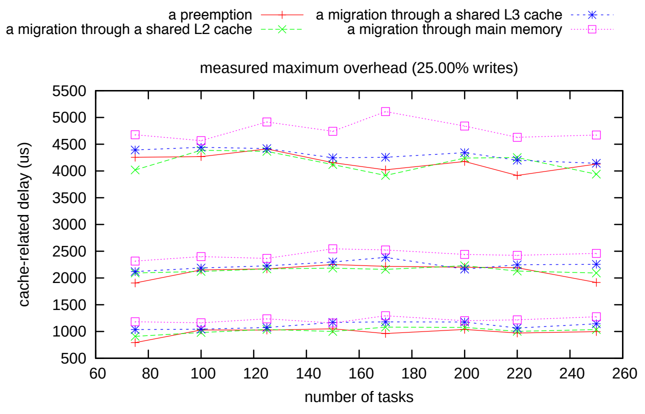


Figure 4: Worst-case CPMD approximations as function of TSS in a system under load (obtained with the schedule-sensitive method). Lines are grouped by WSS: from top: WSS = 1024 KB, WSS = 512 KB, WSS = 256 KB.

Interpretation. The setup used in the experiments depicted in Fig. 2(a,c) simulate *worst-case scenarios* in which a job is preempted by a higher-priority job with a large WSS that (almost) completely evicts the preempted job's WS while activity on other processors generates significant memory bus contention. In contrast, insets (b,d) correspond to situations in which the preempting job does not cause many evictions (which is the case if it has a virtually empty WS or its WS is already cached) and the rest of the system is idle, *i.e.*, insets (b,d) depict *best-case scenarios*. Hence, Fig. 2(a) (resp., Fig. 2(c)) shows the observed worst-case (resp., average) cost of reestablishing cache affinity in a worst-case situation, whereas Fig. 2(b) (resp., Fig. 2(d)) shows the worst-case (resp., average) cost of reestablishing cache affinity in a best-case situation.

Further note that, even though the synthetic method relies on a background workload to generate memory bus contention, the data shown in Fig. 2(a,c) also applies to scenarios in which the background workload is absent if the real-time workload itself generates significant memory bus contention.

This has profound implications for empirical comparisons of schedulers. If it is possible that a job's WS is completely evicted by an "unlucky" preemption, then this (possibly unlikely) event must be reflected in the employed schedulability test(s). Thus, unless it can be shown (or assumed) that *all* tasks have only small WSSs and there is *no* background workload (including background OS activity), then bounds on CPMD should be estimated based on the high-contention scenario depicted in Fig. 2(a,c).

Therefore, based on our data, it is *not warranted* to consider migrations to be more costly than preemptions when making worst-case assumptions (*e.g.*, when applying hard real-time schedulability tests). Further, unless memory bus contention is guaranteed to be absent, this is the case even when using average case overheads (*e.g.*, when applying soft real-time schedulability tests).

5 Impact on Schedulability

The *schedulability* of an algorithm with respect to a given scenario is the fraction of task sets that can be shown to meet their timing constraints. It estimates the probability that a randomly chosen task set can be scheduled and is a commonly employed method to compare scheduling algorithms. For example, Baker [3] and Bertogna *et al.* [7] studied schedulability as a function of system load to assess various global and partitioned schedulers. As argued in Sec. 1, schedulability studies should account for CPMD. However, given that CPMD strongly depends on the WSS, assuming any specific value for D_c introduces a bias on the corresponding specific WSS. In this section, we describe a method to integrate CPMD bounds into schedulability studies that overcomes this limitation. We start by discussing the setup previously used in [8, 9, 10, 14], which in turn is based on an earlier design by Baker [3], and then present the corresponding WSS-agnostic extension.

Task set generation. A schedulability study is based on a parametrized task set generation procedure. Said procedure is used to repeatedly create (and test) task sets while varying the parameters over their respective domains. This enables the schedulability under each of the tested algorithms to be evaluated as a function of the task set generation procedure's parameters.

Recall that a task T_i is defined by its WCET e_i and period p_i. A task's *utilization* $u_i = e_i/p_i$ reflects its required processor share; a task set's *total utilization* is given by $\sum_i u_i$. Our generation procedure depends on three parameters: a probability distribution for choosing u_i, a probability distribution for choosing p_i, and a *utilization cap* U.

Tasks are created by choosing u_i and p_i from their respective distributions and computing e_i. A task set is generated by creating tasks until the total utilization exceeds U and by then discarding the last-added task (unless U is reached exactly). Discarding the last task ensures that all parameters stem from their respective distributions, but the total utilization of the resulting task set may be less than U. Alternatively, the last-added task's utilization can be scaled such that U is reached exactly. This procedure lends itself to studying schedulability as a function of system load by varying U from zero to m while assuming fixed choices for utilization and period distributions (*e.g.*, see [8, 9]).

Accounting for overheads. Task execution costs are commonly inflated to accomodate overheads caused by scheduling decisions, context switches, timer ticks, job releases, and other OS activity during the execution of one job [8, 9, 10, 14, 28]. Such system overheads must be accounted for after a task set has been generated, since most overheads are TSS-dependent [8, 9, 14].

This is in stark contrast to CPMD, which our experiments revealed to be independent of TSS, as discussed in Sec. 4 (Observation 5). Instead, bounding CPMD requires knowledge of a task's WSS. Thus, either a specific WSS must be assumed throughout the study, or a WSS must be chosen randomly during task set generation. Anticipating realistic WSS distributions is a non-trivial challenge, hence prior studies [8, 9, 14] focused on selected WSSs.

Implicit WSS. Instead, CPMD should be an additional parameter of the task set generation procedure, thus removing the need for WSS assumptions. In this *WSS-agnostic setup*, schedulability (*i.e.*, the ratio of task sets deemed schedulable for given parameters) is a function of two variables (U and D_c) and can therefore be studied assuming a wide range of values for D_c (and thus WSS).

While conceptually simple and appealing due to the avoidance of a WSS bias, this setup poses some practical problems. Besides squaring the number of required samples, a "literal" plotting of the results requires a 3D projection, which renders the results virtually impossible to interpret (schedulability plots routinely show four to eight individual curves, *e.g.*, [3, 8, 9]). To overcome this, we propose the following aggregate performance metric instead.

Weighted schedulability. Let $S(U, D_c) \in [0, 1]$ denote the schedulability for a given U and D_c under the WSS-agnostic setup, and let Q denote a set of evenly-spaced utilization caps (*e.g.*, $Q = \{1.0, 1.1, 1.2, \ldots, m\}$). Then *weighted schedulability* $W(D_c)$ is defined as

$$W(D_c) = \frac{\sum_{U \in Q} U \cdot S(U, D_c)}{\sum_{U \in Q} U}.$$

This metric reduces the obtained results to a two-dimensional (and thus easier to interpret) plot without introducing a fixed utilization cap. Weighting individual schedulability results by U reflects the intuition that high-utilization task systems have higher "value" since they are more difficult to schedule. Note that $W(0) = 1$ for an optimal scheduler (if other overheads are negligible).

Weighted schedulability offers the great benefit of clearly exposing the *range of CPMDs* in which a particular scheduler is competitive. Recall from Sec. 1 that global schedulers are provably superior if CPMD is negligible, but not so if migrations are costly. At which point does partitioning become the superior choice? This can be inferred from the weighted schedulability, as is demonstrated next.

Example. Fig. 5 shows $W(D_c)$ assuming soft timing constraints for four schedulers on our 24-core experimental platform: G-EDF, P-EDF, and two C-EDF configurations with clusters of two (resp., six) processors each chosen based on L2 (resp., L3) cache sharing. Task parameters are uniformly distributed, with $u_i \in [0.5, 0.9]$ and $p_i \in [3\text{ms}, 33\text{ms}]$. This workload is of particular interest since smooth video playback and interactive games fall in this range of periods, and high per-task utilizations can easily result from high-definition multimedia processing.

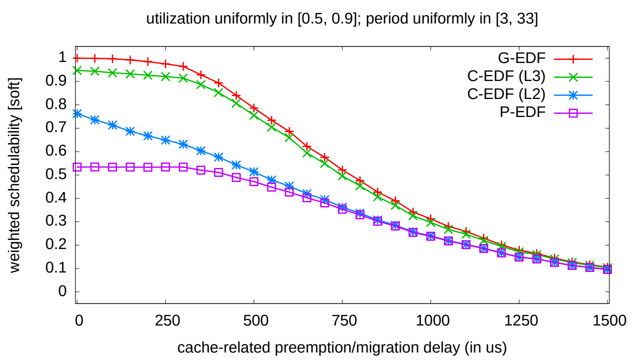


Figure 5: Weighted soft schedulability as a function of CPMD.

Fig. 5 clearly reveals the tradeoff between bin-packing limitations under P-EDF and migration costs under G-EDF. Let D_c^G (resp., D_c^P) denote a bound on CPMD under G-EDF (resp., P-EDF), and let W^G (resp., W^P) denote weighted schedulability under G-EDF (resp., P-EDF). In Fig. 5, the G-EDF curve dominates the P-EDF curve. This implies that G-EDF is a superior choice assuming equal CPMD *i.e.*, if $D_c^G = D_c^P$ then $W^G(D_c^G) \geq W^P(D_c^P)$.

More interesting is the case in which migrations are costly, *i.e.*, $D_c^G > D_c^P$. Suppose preemptions are negligible, *e.g.*, $D_c^P = 0\mu s$: at which point does P-EDF become preferable to G-EDF? The curve for P-EDF reveals that $W^P(0) \approx 0.55$; by tracing G-EDF's curve we find that weighted schedulability under G-EDF drops below 0.55 at $D_c^G \approx 725\mu s$. Thus, G-EDF is preferable to P-EDF, *i.e.*, $W^G(D_c^G) > W^P(D_c^P)$, if $D_c^G < 725\mu s$.

These results should be combined with the actual observed CPMD: under P-EDF, jobs incur only preemptions, whereas they may incur both migrations and preemptions under G-EDF. In a system under load, Fig. 2(a,c) reveals that $D_c^G \approx D_c^P$ in both the average and the worst case, and thus G-EDF is preferable for any WSS. In contrast, in an idle system, $D_c^G > D_c^P$, but, as shown in Fig. 2(b,d), D_c^G does not exceed $725\mu s$ for WSSs smaller than 1024 KB, and thus G-EDF is preferable for such WSSs.

This illustrates that weighted schedulability, in combination with actual CPMD measurements, can reveal interesting tradeoffs between schedulers that cannot be inferred from overhead-oblivious schedulability studies.

6 Conclusion

We have presented two methods for measuring CPMD: the schedule-sensitive method can detect scheduler-dependent cache-related delays since it does not alter the scheduling policy, while the synthetic method rapidly produces large numbers of samples by artificially triggering preemptions and migrations. We have discussed strengths and weaknesses of the two approaches, and have demonstrated their efficacy by reporting average and maximum CPMD for various WSSs on a 24-core Intel UMA machine with two layers of shared caches. Our findings show that, on our platform, CPMD in a system under load is only predictable for WSSs that do not thrash the L2 cache. We further observed that preemption and migration delays did not differ significantly under load, which calls into question the widespread belief that migrations are necessarily more costly than preemptions. In particular, our data indicates that (on our platform) preemptions and migrations differ only little in terms of both worst-case and average-case CPMD if cache affinity is lost completely in the presence of either a background workload or other real-time tasks with large WSSs. Additionally, our experiments showed that incurred CPMD depends on preemption length, but not on task set size.

We have further proposed a method for incorporating CPMD bounds into large-scale schedulability studies without biasing results towards a particular WSS choice. Based on weighted schedulability, this method allows regions to be identified in which a particular scheduler is competitive.

Limitations. Since our methods are based on empirical measurements, they cannot be used to derive safe bounds on true worst-case delays. Further, our current implementation focuses on data caches (but could be extended to apply to instruction caches) and cannot detect if samples were disturbed by the processing of non-maskable interrupts. Nonetheless, we believe that our approach offers a good tradeoff between experimental complexity and accuracy, and hope that it will enable CPMD to routinely be considered in future evaluations of multiprocessor schedulers.

Future work. We plan to validate our experiments by substituting the TSC with performance counters to directly measure cache misses. Further, we would like to apply our measurement methodology to embedded and NUMA platforms. Repeating these experiments in the presence of frequent DMA transfers by I/O devices and atomic (*i.e.*, bus-locking) instructions could yield further insights. Based on the observed trends, further research into bounds on maximum per-task preemption lengths and non-preemptive global schedulers is warranted.

Acknowledgement. We thank Alex Mills for his valuable and helpful suggestions regarding the data analysis.

A CPMD Data

The CPMD data corresponding to the graphs shown in Fig. 2 is given in Tables 1–4.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208, 2005.

[3] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, Florida State University, 2005.

[4] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*. Chapman Hall/CRC, 2007.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[6] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1994.

[7] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, 2005.

[8] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224, 2009.

[9] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169, 2008.

[10] B. Brandenburg, H. Leontyev, and J. Anderson. Accounting for interrupts in multiprocessor real-time systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 273–283, 2009.

[11] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 204–219, 1996.

[12] J. Calandrino. *On the Design and Implementation of a Cache-Aware Soft Real-Time Scheduler for Multicore Platforms*. PhD thesis, University of North Carolina at Chapel Hill, 2009.

[13] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256, 2007.

[14] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, 2006.

[15] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

[16] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[17] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 101–110, 2006.

[18] F. M. David, J. C. Carlyle, and R. H. Campbell. Context switch overheads for Linux on ARM platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[19] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, North Carolina, 2006.

[20] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the 7th ACM International Conference on Embedded Software*, pages 245–254, 2009.

WSS (KB)	Preemption	Migrat. through L2	Migrat. through L3	Migrat. through Mem.
4	17.52	19.09	18.94	21.58
8	35.98	32.89	35.48	32.55
16	69.76	76.13	69.73	61.71
32	136.16	147.49	159.10	137.55
64	248.86	248.82	252.63	244.07
128	525.08	520.77	484.50	520.55
256	1,027.77	1,020.08	1,031.80	1,088.35
512	2,073.41	2,064.59	1,914.32	2,333.64
1,024	3,485.44	4,241.11	4,408.33	3,935.43
2,048	7,559.04	7,656.31	8,256.06	8,375.53
3,072	9,816.22	10,604.52	9,968.44	12,491.07
4,096	12,936.70	14,948.87	12,635.93	15,078.12
8,192	26,577.31	25,760.44	24,923.14	26,091.24
12,288	37,139.30	39,559.55	36,923.48	36,688.75

Table 1: CPMD data. Worst-case delay (in μs) in a system under load. This table corresponds to Fig. 2(a).

WSS (KB)	Preemption	Migrat. through L2	Migrat. through L3	Migrat. through Mem.
4	0.49	3.38	4.27	3.98
8	0.45	5.99	8.08	7.27
16	0.65	11.22	15.53	13.50
32	0.79	17.28	31.01	26.10
64	0.97	17.15	60.91	51.33
128	1.10	14.95	120.47	98.25
256	19.05	30.60	241.68	199.54
512	11.46	17.88	481.52	397.67
1024	30.03	52.63	935.29	784.89
2048	239.45	235.94	1,819.50	1,567.65
3072	567.54	713.33	2,675.71	2,287.87
4096	283.65	288.22	1,523.32	1,169.90
8192	60.23	47.90	522.20	606.40
12288	107.68	109.94	472.15	690.81

Table 2: CPMD data. Worst-case delay (in μs) in an idle system. This table corresponds to Fig. 2(b).

[21] D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proceedings of the 17th International Conference on Real-Time and Network Systems*, pages 45–54, Paris France, 2009.

[22] L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *Proceedings of the 2007 Conference on Design, Automation and Test in Europe*, pages 1623–1628, 2007.

[23] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, 2001.

[24] H. Leontyev. *Compositional Analysis Techniques For Multiprocessor Soft Real-Time Scheduling*. PhD thesis, University of North Carolina at Chapel Hill, 2010.

[25] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[26] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, January 1973.

[27] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker. Characterizing and modeling the behavior of context switch misses. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–101, 2008.

[28] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[29] J. C. Mogul and A. Borg. The effect of context switches on cache performance. *ACM SIGPLAN Notices*, 26(4):75–84, 1991.

[30] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 201–206, 2003.

[31] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, to appear, 2008.

[32] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 80–89, 2009.

	Preemption		Migrat. through L2		Migrat. through L3		Migrat. through Mem.	
WSS (KB)	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
4	5.24	2.31	5.37	2.36	5.66	2.07	5.77	2.08
8	9.14	4.18	9.24	4.21	9.93	3.69	10.09	3.82
16	17.02	8.04	17.05	7.77	18.58	7.00	18.78	7.18
32	32.88	15.94	32.93	15.73	35.82	13.96	35.99	14.30
64	65.05	31.38	65.33	31.90	70.72	27.87	70.50	28.02
128	128.64	62.08	127.09	61.22	137.35	52.48	141.05	58.20
256	248.81	117.10	246.34	119.89	267.73	106.19	272.56	115.37
512	478.45	239.08	476.95	251.41	507.27	227.87	509.18	245.06
1024	739.20	515.18	733.27	624.26	772.68	544.80	810.37	641.08
2048	740.10	1,200.93	773.22	1,409.55	837.53	1,373.93	853.27	1,605.60
3072	355.76	1,781.11	400.88	2,021.39	377.96	1,974.79	483.20	2,373.09
4096	247.88	2,456.97	291.93	2,756.90	274.51	2,622.08	350.07	3,118.26
8192	212.90	4,793.77	374.45	5,230.35	436.19	5,153.05	282.28	5,797.23
12288	201.20	7,211.18	333.80	7,683.50	467.50	7,485.35	274.23	8,122.10

Table 3: CPMD data. Average-case delay (in μs) in a system under load. This table corresponds to Fig. 2(c).

	Preemption		Migrat. through L2		Migrat. through L3		Migrat. through Mem.	
WSS (KB)	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
4	0.13	0.06	2.91	0.18	3.98	0.06	3.48	0.12
8	0.11	0.06	5.31	0.39	7.75	0.07	6.54	0.15
16	0.17	0.09	10.19	0.72	15.17	0.10	12.62	0.24
32	0.26	0.10	14.41	1.24	30.11	0.14	24.82	0.38
64	0.29	0.15	14.21	1.29	59.79	0.22	49.18	0.72
128	-0.17	0.30	12.89	1.08	118.23	0.47	94.60	1.40
256	0.83	0.45	15.02	1.14	236.94	1.43	192.42	2.88
512	1.62	0.66	13.96	1.30	477.22	1.69	384.35	4.48
1024	3.85	1.45	18.28	1.70	921.61	4.79	769.80	7.93
2048	26.21	15.80	45.61	16.03	1,721.14	130.92	1,459.52	120.58
3072	74.04	42.28	104.26	49.47	1,867.61	246.62	1,501.23	229.21
4096	39.66	23.96	51.29	24.75	1,156.36	153.95	790.41	157.98
8192	1.60	9.06	0.70	9.60	468.26	14.14	482.73	66.05
12288	5.84	13.85	6.62	12.16	385.62	24.14	420.76	57.21

Table 4: CPMD data. Average-case delay (in μs) in an idle system. This table corresponds to Fig. 2(d).

[33] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

[34] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[35] G. Stamatescu, M. Deubzer, J. Mottok, and D. Popescu. Migration overhead in multiprocessor scheduling. In *Proceedings of the 2nd Embedded Software Engineering Conference*, pages 645–654, 2009.

[36] J. Stärner and L. Asplund. Measuring the cache interference cost in preemptive real-time systems. *ACM SIGPLAN Notices*, 39(7):146–154, 2004.

[37] J. Staschulat and R. Ernst. Scalable precision cache analysis for real-time software. *ACM Transactions on Embedded Computing Systems*, 6(4):25, 2007.

[38] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303, 2008.

[39] D. Thiebaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[40] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[41] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *Performance Evaluation: Origins and Directions (LNCS 1769)*, pages 97–139, 2000.

[42] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[43] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.